Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»

**Отчет**
По лабораторной работе №3
По курсу «Конструирование компиляторов»
На тему
«Синтаксический разбор с использованием метода рекурсивного спуска»

Студент: Мурашов И.Д.
Группа: ИУ7-23М
Вариант: 1
Преподаватель: Ступников А.А.

Москва, 2020

# Оглавление

# 1 Цель и задачи работы

**Цель работы**: приобретение практических навыков реализации синтаксический разбор с использованием метода рекурсивного спуска.

<выражение> -> <простое выражение> | <простое выражение> <операция отношения> <простое выражение>
<простое выражение> -> <терм> | <знак> <терм> | <простое выражение> <операция типа сложения> <терм>
<терм> -> <фактор> | <терм> <операция типа умножения> <фактор>
<фактор> -> <идентификатор> | <константа> | ( < простое выражение > ) | not <фактор>
<операция отношения> -> = | <> | < | <= | > | >=
<знак> -> + | -
<операция типа сложения> -> + | - | or
<операция типа умножения> -> * | / | div | mod | and

Замечания.

1. Нетерминалы <идентификатор> и <константа> - это лексические единицы (лексемы), которые оставлены неопределенными, а при выполнении лабораторной работы можно либо рассматривать их как терминальные символы, либо определить их по своему усмотрению и добавить эти определения.

2. Терминалы not, or, div, mod, and - ключевые слова (зарезервированные).

3. Терминалы ( ) - это разделители и символы пунктуации.

4. Терминалы = <> < <= > >= + - * / - это знаки операций.

5. Нетерминал <выражение> - это начальный символ грамматики.

Данная грамматика дополнятеся блоком, состоящим из последовательности операторов присваивания в стиле Алгол-Паскаль.
<программа> -> <блок>
<блок> -> begin <список операторов> end
<список операторов> -> <оператор> | <список операторов> ; <оператор>
<оператор> -> <идентификатор> = <выражение>

## 2 Текст программы

```
using Lab3_Descending.Elements;
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace Lab3_Descending
{
public class Program
{
public static void Main(string[] args)
{

Dictionary<NonTerminal, List<Generation>> productions = XmlWorker.Gra
out List<NonTerminal> nonTerminals, out NonTerminal start);
SaveReadableFormat("grammar_in.txt", terminals, nonTerminals, product

string input = "";
while (true)
{
Console.Write("Input string: ");
input = Console.ReadLine();
if (input == "exit")
break;

input = input.Replace(" ", string.Empty);

if (RecAddProc.Parse(input))
Console.WriteLine("Ok!\n");
else
Console.WriteLine("Not ok!\n");
}
}

private static void SaveReadableFormat(string filename, List<Terminal
Dictionary<NonTerminal, List<Generation>> productions)
{
StringBuilder terms = new StringBuilder("T: ");
foreach (var terminal in terminals)
terms.Append($"{terminal.Spell}, ");

StringBuilder nonterms = new StringBuilder("NT: ");
string start = "";
foreach (var nonTerminal in nonTerminals)
{
nonterms.Append($"{nonTerminal.Name}, ");
if (nonTerminal.IsStart)
start = nonTerminal.Name;
```

```
}

List<StringBuilder> sProductions = new List<StringBuilder>();
foreach (var produciton in productions)
{
StringBuilder sProduction = new StringBuilder($"{produciton.Key.Name}
foreach (var generation in produciton.Value)
{
sProduction.Append(generation.GenString);
sProduction.Append(" | ");
}
sProductions.Add(sProduction);
}

using (StreamWriter sw = new StreamWriter(filename, false))
{
sw.WriteLine(terms.ToString());
sw.WriteLine(nonterms.ToString());
foreach (var sProduction in sProductions)
sw.WriteLine(sProduction.ToString());
sw.WriteLine(start);
}
}
}
}
using System;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Text;
using System.Transactions;

namespace Lab3_Descending
{
class RecAddProc
{

private static int cur;
private static string curString;
private static string errMsg = "";

public static bool Parse(string input)
{
cur = 0;
curString = input;

bool result = true;
if (!PROG(curString))
{
result = false;
}
```

```
if (cur != input.Length)
{
result = false;
Console.WriteLine("Something else after statement!");
}
return result;
}


public static bool MOP(string input)
{

if (curString.StartsWith("*"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("/"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("div"))
{
cur += 3;
curString = curString.Substring(3, curString.Length - 3);
}
else if (curString.StartsWith("mod"))
{
cur += 3;
curString = curString.Substring(3, curString.Length - 3);
}
else if (curString.StartsWith("and"))
{
cur += 3;
curString = curString.Substring(3, curString.Length - 3);
}
else
{
//errMsg = $"Expected boolean const at position {cur}";
//Console.WriteLine(errMsg);
return false;
}

return true;
}

public static bool SIGN(string input)
{
```

```
if (curString.StartsWith("+"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("-"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else
{
//errMsg = $"Expected boolean const at position {cur}";
//Console.WriteLine(errMsg);
return false;
}

return true;
}

public static bool AOP(string input)
{

if (curString.StartsWith("+"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("-"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("or"))
{
cur += 2;
curString = curString.Substring(2, curString.Length - 2);
}
else
{
//errMsg = $"Expected boolean const at position {cur}";
//Console.WriteLine(errMsg);
return false;
}

return true;
}

public static bool ROP(string input)
{
```

```
if (curString.StartsWith("="))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("<>"))
{
cur += 2;
curString = curString.Substring(2, curString.Length - 2);
}
else if (curString.StartsWith("<"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith("<="))
{
cur += 2;
curString = curString.Substring(2, curString.Length - 2);
}
else if (curString.StartsWith(">"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
}
else if (curString.StartsWith(">="))
{
cur += 2;
curString = curString.Substring(2, curString.Length - 2);
}
else
{
//errMsg = $"Expected boolean const at position {cur}";
//Console.WriteLine(errMsg);
return false;
}

return true;
}

public static bool F(string input)
{

if (curString.StartsWith("a"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
return true;
}
```

```
else if (curString.StartsWith("c"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
return true;
}
else if (curString.StartsWith("("))
{
var tmpcur = cur;
var tmpcurStriong = curString;

cur += 1;
curString = curString.Substring(1, curString.Length - 1);

if (SEXP(input))
{
if (curString.StartsWith(")"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
return true;
}
return false;
}
else
return false;

}
else if (curString.StartsWith("not"))
{
cur += 3;
curString = curString.Substring(3, curString.Length - 3);

if (F(input))
{
return true;
}
else
return false;

}
else
{
//errMsg = $"Expected boolean const at position {cur}";
//Console.WriteLine(errMsg);
return false;
}
}
```

```
public static bool EXP(string input)
{
bool res = false;

var tmpcur = cur;
var tmpcurString = curString;

if (SEXP(input))
{
if (ROP(input))
{
if (SEXP(input))
{
res = true;
}
}
}

if (res != true)
{
cur = tmpcur;
curString = tmpcurString;

if (SEXP(input))
{
res = true;
}
}

if (res != true)
{
cur = tmpcur;
curString = tmpcurString;
}

return res;
}

private static bool SEXP(string input)
{
var res = false;
var tmpcur = cur;
var tmpcurString = curString;

if (T(input))
{
if (SEXP1(input))
{
res = true;
}
```

```
}

if (res != true)
{
cur = tmpcur;
curString = tmpcurString;
if (SIGN(input))
{
if (T(input))
{
if (SEXP1(input))
{
res = true;
}
}
}
}

if (res != true){
cur = tmpcur;
curString = tmpcurString;
}

return res;
}

private static bool SEXP1(string input)
{
var res = false;
var tmpcur = cur;
var tmpcurString = curString;

if (AOP(input))
{
if (T(input))
{
if (SEXP1(input))
{
res = true;
}
}
}
// EPS
else
{
res = true;
cur = tmpcur;
curString = tmpcurString;
}
```

```
if (res != true)
{
cur = tmpcur;
curString = tmpcurString;
}

return res;


}

private static bool T(string input)
{
var res = false;
var tmpcur = cur;
var tmpcurString = curString;

if (F(input))
{
if (T1(input))
{
res = true;
}
}

if (res != true) {
cur = tmpcur;
curString = tmpcurString;
}

return res;
}

private static bool T1(string input)
{
var res = false;
var tmpcur = cur;
var tmpcurString = curString;

if (MOP(input))
{
if (F(input))
{
if (T1(input))
{
res = true;
}
}
}
}
// EPS
```

```
else
{
res = true;
cur = tmpcur;
curString = tmpcurString;
}

if (res != true) {
cur = tmpcur;
curString = tmpcurString;
}



return res;

}


public static bool PROG(string input)
{
bool result = true;
if (!BLOCK(input))
{
result = false;
}
return result;
}

public static bool BLOCK(string input)
{
bool result = true;
if (curString.StartsWith("begin"))
{
cur += 5;
curString = curString.Substring(5, curString.Length - 5);
if (OPLIST(input))
{
if (curString.StartsWith("end"))
{
cur += 3;
curString = curString.Substring(3, curString.Length - 3);
}
else
{
result = false;
}
}
}
else
```

```
{
result = false;
}
return result;
}

public static bool OPLIST(string input)
{
bool result = true;
int savedCur = cur;
string savedInput = input;

if (OPERATOR(input))
{
if (!OPLIST1(input))
{
result = false;
}
}
else
{
result = false;
}

return result;
}

public static bool OPLIST1(string input)
{
bool result = true;
if (curString.StartsWith(";"))
{
cur++;
curString = curString.Substring(1, curString.Length - 1);
if (OPERATOR(input))
{
if (!OPLIST1(input))
{
result = false;
}
}
else
{
result = false;
}
}

return result;
}
```

```
public static bool OPERATOR(string input)
{
bool result = true;
int savedCur = cur;
string savedInput = curString;

if (curString.StartsWith("a"))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
if (curString.StartsWith("="))
{
cur += 1;
curString = curString.Substring(1, curString.Length - 1);
if (!EXP(input))
{
result = false;
}
}
}
else
{
result = false;
}


return result;
}

}
}
```

# 3 Проверка правильности программы

**Тестовые данные**:

| Входная строка | Ожидаемый результат | Результат |
|---|---|---|
| begina=aend | True | True |
| begina=a;a=c<aend | True | True |
| beginend | False | False |
| beginaend | False | False |
| begina;end | False | False |
| begina=c;end | False | False |
| begina=a/end | False | False |

# 4 Выводы

По результатам проведенной работы были приобретены практические навыки реализации синтаксического разбора с использованием метода рекурсивного спуска. Также была раз-

работана, отестирована отлажана программа реализующая метод рекурсивного спуска.

# 5 Список литературы

1. АХО А.б УЛЬМАН Дж. Теория синтакстического анализа, перевода и компиляции: В 2-х томах. Т1.: Синтаксический анализ. - М.: Мир, 1978.
2. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.